THE UNIVERSITY
of MANCHESTER

February 18, 2000

Computer Science
University of Manchester

Even faster code trees

Alexandre Riazanov and Andrei Voronkov

Department of Computer Science
University of Manchester
Preprint Series
CSPP-5

Even faster code trees

Alexandre Riazanov* Andrei Voronkov†

February 18, 2000

Abstract

Code trees [Voronkov 1995] is an indexing technique used for implementing several indexed operations on terms in the theorem prover Vampire [Ryazanov and Voronkov 1999]. Code trees offer greater flexibility than discrimination trees. In this paper we review a new, considerably faster, version of code trees based on a different representation of the query term. We also introduce a partially adaptive version of code trees.

Recent preprints issued by the Department of Computer Science, Manchester University, are available on WWW via URL `http://www.cs.man.ac.uk/preprints/index.html` or by ftp from `ftp.cs.man.ac.uk` in the directory `pub/preprints`.

---

*Department of Computer Science, University of Manchester. Partially supported by an EPSRC grant, ORS award, and a grant from the Faculty of Science of Technology.

†Department of Computer Science, University of Manchester. Partially supported by an EPSRC grant and a grant from the Faculty of Science and Technology.

Contents

1 Introduction

In [Voronkov 1995] code trees, a new indexing technique for forward subsumption, was presented. In order to implement efficiently forward subsumption on a large set of clauses a general subsumption algorithm is specialised at run time for each particular clause in the set. The specialised version of the algorithm is represented as a sequence of instructions of some abstract machine. Such codes are integrated into an indexing structure — a code tree, which allows one to perform subsumption check by the whole set of clauses at once. Although code trees can be considered as a differently presented version of discrimination trees, the compilation-based approach gives some serious advantages. Code sequences for indexed terms are rather flexible objects as they allow various equivalence-preserving transformations to be performed on the index. This flexibility enables invention and formulation of new optimisations. Exploiting the notion of abstract machine makes description of the indexed technique more machine-oriented and its efficient implementation feasible.

Although experiments with the original version of code trees have shown high effectiveness of the compilation-based approach, a case study revealed that the original formulation of this technique leaves space for significant improvements. In this report we discuss several improvements implemented in version 0.0 of *Vampire* [Ryazanov and Voronkov 1999] that has won CASC'16 [Sutcliffe 2000] in the mixed category.

The main improvement was achieved by changing the representation of query clauses. The original version [Voronkov 1995] deals with query terms represented as tree-like structures. It has been discovered that the *flatterm* [Christian 1993] representation of query clauses eliminate the need in some operations in code trees and also makes expensive operations of term comparison faster. We will describe the new version of code trees in Section 3.

Apart from the representation of queries, another shortcoming in the original version of code trees is worth special attention. There are two factors that can increase the efficiency of indexing techniques: *early detection of failure* and *better sharing* of structure (or in our case sharing of code). In the case of code trees early detection of failure can be achieved by applying term comparison instructions as early as possible. At the same time this can deteriorate sharing to a very high extent, so that the size of a code tree grows as much as 10 times on some benchmarks. In Section 4 we describe a partially adaptive version of code trees in which both early detection of failure and better code sharing is achieved by moving term comparison instructions up and down the tree during the compilation of indexed terms. Moreover, we can change the comparison instructions to achive better sharing. We call the resulting version of code trees *partially adaptive* because the tree can adapt to insertion of new instructions by changing itself. The ability to partially adapt code trees with small overheard shows their advantage over the more standard data structures used for forward subsumption and similar clause retrieval operations, for example discrimination trees [McCune 1992].

In Section 5 we briefly describe other optimizations we tried to perform on code trees by adding more instructions intended at early detection of failure. Finally, in Section 6 we describe experiments with partially adaptive code trees.

As a case study, we discuss forward subsumption. However, code trees are used in Vampire also for forward simplification by unit equalities and in some other simplification algorithms.

2 Preliminaries

We assume acquaintance with the basic notions of terms, substitutions and clauses. A clause C_1 *subsumes* a clause C_2 if there exists a substitution θ such that $C_1\theta$ is a subset of C_2. In [Voronkov 1995] indexing for multiliteral clauses was done by composition of indexes for their literals. Since our current approach to dealing with multiliteral clauses does not differ from the one of [Voronkov 1995], in this report we consider only *unit clauses*, i.e. clauses with one literal. Our main optimizations can be easily illustrated already for the unit clause case. In the case of unit clauses, subsumptions can be reformulated as the matching problem on terms. We say that a term t_2 *matches* a term t_1 if there exists a substitution θ such that $t_1\theta = t_2$. In this case we will also say that t_1 *subsumes* t_2.

We will follow the general framework of term indexing presented in [Ramakrishnan, Sekar and Voronkov 2000]. In general, the term indexing problem can be formulated as follows. Given a set of terms I, called the set of *indexed terms* and a single term t, called the *query term*, we have to quickly retrieve each term $s \in I$ such that a *retrieval condition* R holds between s and t, i.e. we have $R(s,t)$. For the purpose of this report the retrieval condition R is *forward subsumption*: $R(s,t)$ holds if s subsumes t. The *term indexing problem* consists of funding a datastructure, called the *index* which allows one to perform efficiently the following operations: *term retrieval*, i.e. finding all (or some) $s \in I$ that are in relation R with the query term t, and *index maintenance*: changing the index when terms are inserted into or deleted from, the set of indexed term.

Forward subsumption is one of the most important procedures in first-order theorem provers, on some benchmarks it consumes most of the time. Some other important operations in first-order theorem proving are based on essentially the same retrieval condition as forward subsumption, for example forward simplication by unit equalities. The main differences between forward subsumption and forward simplification by unit clauses is that the former only requires to check whether there exists $s \in I$ subsuming t while the latter requires to find all such s together with a matching substitution θ such that $s\theta = t$.

A *code tree* is a datastructure for term indexing. The main idea of code trees is as follows. Let F be a procedure for performing forward subsumption, so $F(s,t)$ returns true is s subsumes t. For each indexed term $s \in I$ we specialize F by fixing its first argument to s. This specialized procedure is denoted by F_s, thus we have $F_s(t) = F(s,t)$ for all terms s and t. The procedure F_s for each indexed term $s \in I$ is represented as a sequence of instructions of an *abstract subsumption machine*. There is a small number of instructions, some of them have parameters. Then the procedures $\{F_s \mid s \in I\}$ are combined into a larger set of instructions F_I, called the *code tree for I*. The set of instructions F_I is better viewed as a tree rather than a sequence, hence the name *code tree*. The set of instructions F_I is a procedure that can be executed on any query term t such that $F_I(t) \leftrightarrow (\exists s \in I)F_s(t)$.

3 Code trees for the flatterm-based representation of query terms

The title of this report is ambiguous. Indeed, what is even faster: code in code trees or code trees themselves? In this section we show that a new representation of query terms gives much faster and compact code compared to the original version [Voronkov 1995]. In Section 4 we show that the trees can also be faster.

In this section we describe a version of code trees obtained by adapting the original one of [Voronkov 1995] to the new representation of queries. Following [Voronkov 1995], we start from considering compilation of terms for the case of forward subsumption by one clause. To represent our algorithms formally, we will need quite a few definitions.

For simplicity, we start with the following problem: given an indexed term t and a query term q we have to check if t subsumes q, and then consider the case of many indexed terms.

The algorithm traverses the query term in the left-to-right depth-first order. To formalize this order of traversal, in Section 3.1 we introduce notions of position in a term and order $<$ on such positions. We also introduce two operations on positions *next* and *after*. The operation *next* finds the next position for such a traversal, the operation *after* finds the position after the positions of all subterms of the current subterm. During the traversal of the query term we check that function symbols in it coincide with the corresponding function symbols of the indexed term, so we introduce a function *top* that returns the outermost symbol of a term. In addition, during traversal of the query term our algorithms memorizes subterms of the query term q corresponding to the variable positions in the indexed term t. For example, if the indexed term is $f(x_1, x_1, x_2, x_1, x_2, a)$ and the query term has the form $f(t_1, t_2, t_3, t_4, t_5, a)$, then the algorithm memorizes the terms t_1–t_5. In order to check that the query term is subsumed by the indexed term, we have to verify that $t_1 = t_2 = t_4$ and $t_3 = t_5$. Note that the verification can be done in several ways, for example we can compare t_1 with t_2 first, t_3 with t_5 next, and then t_1 with t_4. Alternatively, we can verify (in this order) $t_2 = t_4$, $t_1 = t_2$, and then $t_3 = t_5$. To formalize this part of the algorithm, we define variable positions in the term. We also define computation sequences that formalize different orders of making term comparisons.

3.1 Positions in term

If t is a term, $top(t)$ denotes the *top symbol* of t defined as follows:

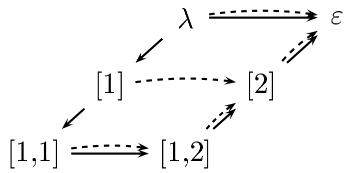

Figure 1: *next* and *after* on the positions in $f(f(a,a),a)$. Solid straight lines and dashed arcs depict *next* and *after* correspondingly.

$$top(t) = \begin{cases} t, & \text{if } t \text{ is a variable or constant;} \\ f, & \text{if } t = f(t_1, \dots, t_n). \end{cases}$$

We call a *position* any finite sequences of natural numbers, including the empty sequence, denoted by λ. The notion of *position in a term* t and the *subterm of* t *at a position* p, denoted t/p, are given by the following definition.

1. the empty position λ is a position in t and $t/p = t$.

2. if $t/p = f(t_1, \dots, t_n), n > 0$, then $p.1, \dots, p.n$ are positions in t and $t/(p.i) = t_i$ for all $i \in \{1, \dots, n\}$.

$Pos(t)$ will denote the set of all positions in t. For technical purposes we we extend $Pos(t)$ by a special object ε called the *end position* in t. The set $Pos(t) \cup \{\varepsilon\}$ will be denoted by $Pos^+(t)$. When it is necessary to tell the end position from other positions, we call the positions from $Pos(t)$ *proper positions*. *Size* of a term t, denoted $|t|$, is defined as the number of proper positions in t. We denote by $<$ the lexicografic ordering on positions extended in the following way: $p < \varepsilon$ for any proper position p. To perform traversal of a term t we will need two operations on proper term positions: $next_t$ and $after_t$, which can be informally explained as follows. Represent the term t as a tree and imagine a term traversal in the left-to-right, depth-first direction. Suppose $t/p = s$. Then $t/next_t(p)$ is the subterm of t visited immediately after s, and $t/after_t(p)$ is the subterm visited immediately after traversal of all subterms of s. Formally, let $\lambda = p_1 < \ldots < p_n < p_{n+1} = \varepsilon$ be all positions in t. Then $next_t(p_i) = p_{i+1}$ for all $i \leq n$. The definition of $after_t$ is as follows: $after_t(\lambda) = \varepsilon$ and for $1 < i < n$ $after_t(p_i) = p_j$, and j is the smallest number such that $j > i$ and for all $i < k < j$ the position p_i is a prefix of p_k. Figure 1 illustrates behaviour of *next* and *after* on the positions in $f(f(a,a),a)$.

As it was mentioned, our new code trees are interpreted on queries represented as *flatterms*. In Vampire we use an *array-based* version of flatterms. A term t is represented by an array of the size $|t| + 1$. Let $p_1 < \ldots < p_n$ be all positions in t. Then the i-th element of the array is a pair $< s, j >$, where $s = top(t/p_i)$ and $p_j = after_t(p_i)$. For example, the term $f(x, g(a))$ is represented by the structure depicted on Figure 2.

In can be seen that computation of our major operations on positions, $next_q$ and $after_q$, can be done very efficiently on such a representation. $next_q$ is computed by a simple incrementation

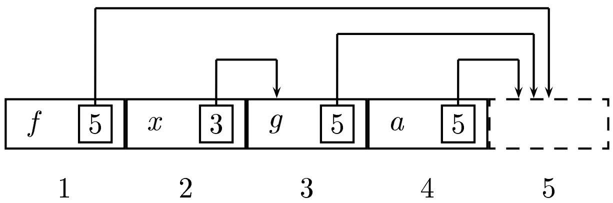

Figure 2: Flatterm structure for $f(x, g(a))$

of the corresponding index, the index of $after_q(p_i)$ is given in the ith element explicitly. Another serious advantage of this representation in comparison with tree-like terms is that equality of two subterms q/p_i and q/p_j can be checked efficiently, without using stack operations.

For technical purposes we introduce a new set of variables $*_1, *_2, \ldots$, called the *technical variables*. A term containing no technical variables will be called an *ordinary term*. Let $\lambda = p_0 < p1 < \ldots < p_n$ be all proper positions in t. For $i \in \{1, \ldots, n\}$, $pos_i(t)$ will denote p_i.

Let $p_{k_1} < \ldots < p_{k_m}$ be all such proper positions in t that t/p_{k_i} is a variable. The *i-th variable position* in t, denoted by $vp_i(t)$, is defined as $vp_i(t) = p_{k_i}$. For $i > m$ $vp_i(t)$ is undefined. The *technical skeleton* of a term t, denoted by $tsk(t)$, is the term obtained from t be replacing the subterm of t at the ith variable position by the technical variable $*_i$, for all i. For example, the technical skeleton of $f(x_1, a, g(x_1, x_2))$ is $f(*_1, a, g(*_2, *_3))$.

The *variable equivalence relation* for a term t, denoted $\mathcal{E}_t$, is the equivalence relation on $\{0, 1, 2, \ldots\}$ such that: $< i, j > \in \mathcal{E}_t$ if and only if $t/vp_i(t) = t/vp_j(t)$. For example, the variable equivalence relation for $f(x_1, a, g(x_1, x_2))$ consists of two equivalence classes: $\{1, 2\}$ and $\{3\}$. The pair $< tsk(t), \mathcal{E}_t >$ will be called the *technical abstraction* of t. Note that two terms have the same technical abstraction if and only if they are variants of each other. If $\mathcal{B}$ is a binary relation, $\mathcal{B}^{\approx}$ denotes the transitive, reflexive and symmetric closure of $\mathcal{B}$. If $\mathcal{E}$ is an equivalence relation and $\mathcal{B}$ is such a binary relation that $\mathcal{B}^{\approx} = \mathcal{E}$, then $\mathcal{B}$ is called a *frame* of $\mathcal{E}$. A frame is called *minimal* if no proper subset of it is a frame. Throughout the rest of the paper we consider only equivalence relations over finite sets of the form $\{1, \ldots, n\}$. A finite sequence $< u_1, v_1 >, \ldots, < u_k, v_k >$ of pairs of integers is called a *computation sequence* for $\mathcal{E}$ if the relation $\{< u_1, v_1 >, \ldots, < u_k, v_k >\}$ is a minimal frame of $\mathcal{E}$ and $u_i < v_i$ for all $i \in \{1, \ldots, k\}$. Such a computation sequence is called *canonical* if each u_i is the minimal element of its equivalence class in $\mathcal{E}$ and for all i, j $v_i < v_j$ if $i < j$. Note that the canonical computation sequence is uniquely defined.

3.2 Compilation for forward subsumption by one clause

We are going to solve the following problem: given a term t and a query term q we have to check if t subsumes q. Figure 3 shows a deterministic algorithm that does the job.

Following [Voronkov 1995] we specialise this general subsumption algorithm *Subsume* for each indexed term t, obtaining its specialized version $Subsume_t$. The specialized version has the property $Subsume_t(q) = Subsume(t, q)$, for each query term q. The specialized algorithm is represented as a sequence of instructions of an abstract machine. In other words, we *compile* the term into code of the abstract machine. Then this code is submitted, together with the query term q, to the interpreting procedure. Before presenting technical details let us consider one simple example.

```
procedure Subsume(t, q)
begin
  /* First phase:  term traversal */
  let subst be an array for storing positions in q;
  pos_t := λ;
  pos_q := λ;
  while pos_t ≠ ε
    if tsk(t)/pos_t = *_i /* t/pos_t is a variable */
      then
        subst[i] := pos_q;
        pos_q := after_q(pos_q);
        pos_t := after_t(pos_t);
      else /* t/pos_t is not a variable */
        if top(t/pos_t) = top(q/pos_q)
          then
            pos_q := next_q(pos_q);
            pos_t := next_t(pos_t);
          else return failure;
        fi;
    fi;
  end while;

  /* Second phase:  comparison of terms */
  let < u_1, v_1 >, ..., < u_n, v_n > be the canonical computation sequence for E_t.
  i := 1;
  while i ≤ n
    if q/subst[u_i] ≠ q/subst[v_i]
      then return failure;
      else i := i + 1;
  end while
  return success;
end
```

Figure 3: A one-to-one subsumption algorithm

procedure $\mathit{Subsume}_t(q)$
begin
$p := \lambda$;
if $\mathit{top}(q/p) \neq f$ **return** failure;
$p := \mathit{next}_q(p)$;
if $\mathit{top}(q/p) \neq g$ **return** failure;
$p := \mathit{next}_q(p)$;
$\mathit{subst}[1] := p$;
$p := \mathit{after}_q(p)$;
$\mathit{subst}[2] := p$;
$p := \mathit{after}_q(p)$;
if $\mathit{top}(q/p) \neq h$ **return** failure;
$p := \mathit{next}_q(p)$;
$\mathit{subst}[3] := p$;
$p := \mathit{after}_q(p)$;
$\mathit{subst}[4] := p$;
$p := \mathit{after}_q(p)$;
if $q/\mathit{subst}[1] \neq q/\mathit{subst}[3]$ **return** failure;
if $q/\mathit{subst}[1] \neq q/\mathit{subst}[4]$ **return** failure;
return success;
end

Figure 4: The algorithm *Subsume* specialized for the term $t = f(g(x_1, x_2), h(x_1, x_1))$

$\mathit{initl} : \mathit{Initialize}(l_1)$
$l_1 : \quad \mathit{Check}(f, l_1, \mathit{faill})$
$l_2 : \quad \mathit{Check}(g, l_3, \mathit{faill})$
$l_3 : \quad \mathit{Put}(1, l_4, \mathit{faill})$
$l_4 : \quad \mathit{Put}(2, l_5, \mathit{faill})$
$l_5 : \quad \mathit{Check}(h, l_6, \mathit{faill})$
$l_6 : \quad \mathit{Put}(3, l_7, \mathit{faill})$
$l_7 : \quad \mathit{Put}(4, l_8, \mathit{faill})$
$l_8 : \quad \mathit{Compare}(1, 3, l_9, \mathit{faill})$
$l_9 : \quad \mathit{Compare}(1, 4, l_{10}, \mathit{faill})$
$l_{10} : \quad \mathit{Success}$
$\mathit{faill} : \mathit{Failure}$

Figure 5: The corresponding sequence of instructions

EXAMPLE 3.1 Let $t = f(g(x_1, x_2), h(x_1, x_1))$ be the compiled term. The specialised version of the matching algorithm for this term is shown in Figure 4.

This specialized version can be rewritten in a more formal way using special instructions *Initialize*, *Check*, *Put*, *Compare*, *Success* and *Failure* as shown in Figure 5. The semantics of these instructions should be clear from the example, but will also be formally explained later.

3.3 Abstract subsumption machine

Now we are ready to describe the abstract machine, its instructions, compilation process, and interpretation formally. Memory of the abstract machine is divided into the following "registers":

1. substitution register *subst* which is an array of positions in the query term;
2. register p for storing the current position in the query term;
3. *instr* for storing the label of the current instruction.

To identify instructions in code we will use special objects — labels. We distinguish two special labels: *initl*, and *faill*. A *labeled instruction* will be written as a pair of the form $l : I$, where l is a label and I is the instruction itself. The instruction set of our abstract machine consists of *Initialize*, *Check*, *Put*, *Compare*, *Success* and *Failure*. *Success* and *Failure* have no arguments. Other instruction have the following form:

Initialize(l_1)	$p := \lambda$; **goto** l_1	*Check*(s, l_1, l_2)	**if** $top(q/p) = s$ **then** $p := next_q(p)$; **goto** l_1 **else** **goto** l_2
Put(n, l_1, l_2)	$subst[n] := p$; $p := after_q(p)$; **goto** l_1	*Compare*(m, n, l_1, l_2)	**if** $q/subst[m] = q/subst[n]$ **then** **goto** l_1; **else** **goto** l_2
Success	**return** success	*Failure*	**return** failure

Figure 6: Semantics of instructions in code sequences

- *Initialize*(l_1), where l_1 is a label;
- *Check*(f, l_1, l_2), where f is a function symbol and l_1, l_2 are labels;
- *Put*(n, l_1, l_2), where n is a positive integer and l_1, l_2 are labels;
- *Compare*(m, n, l_1, l_2), where m, n are positive integers and l_1, l_2 are labels.

For convenience, we define two functions on instructions, *cont* and *back*. On all the above instructions *cont* returns l_1 and *back* returns l_2. Intuitively, *cont* is the label of the instructions that should be executed after the current instruction (if this instruction succeeds), and *back* is the label of the instruction that is executed if the current instruction fails.

The semantics of the instructions is shown in Figure 6. At the moment the last argument of *Put* is dummy. It will be used when we discuss the case of more than one indexed term.

For a given indexed term t, compilation of instructions for $Subsume_t$ results in the set of labeled instructions, called the *code for t*. It consists of two parts: *traversal code* and *compare code* plus three standard instructions: *initl* : *Initialize*(l_1), *succl* : *Success* and *faill* : *Failure*.

Suppose $p_1 < p_2 < \ldots < p_m$ are all positions in t. The *traversal code* for t is the set of instructions $\{l_1 : I_1, \ldots, l_m : I_m\}$, where l_i's are labels and I_i's are defined as follows:

$$I_i = \begin{cases} Check(top(t/p_i), l_{i+1}, faill), & \text{if } t/p_i \text{ is not a variable} \\ Put(k, l_{i+1}, faill), & \text{if } tsk(t)/p_i = *_k \end{cases}$$

Let $< u_1, v_1 >, \ldots, < u_n, v_n >$ be the canonical computation sequence for $\mathcal{E}_t$. Then the *compare code* for t is the set of instructions $l_{m+i} : Compare(u_i, v_i, l_{m+i+1}, faill)$ for $i \in \{1, \ldots, n\}$, where $l_{m+n+1} = succl$. In Figure 5 for Example 3.1 the instructions $l_1 - l_7$ form the traversal code and the instructions l_8, l_9 form the compare code.

The code for t is executed on the query term according to the semantics of instructions shown in Figure 6, beginning with the instruction *Initialize*.

The following statement will unlikely surprise anybody:

THEOREM 3.2 *The execution of the code for t on any query term q terminates. It returns success if and only if t subsumes q.*

Observe that code for t has a linear structure: instructions can be executed sequentially, so that the nth instruction either succeeds and calls the $n+1$th instruction, or fails. In view of this observation we will call code for t also the *code sequence* for t.

3.4 Code trees for many-to-one subsumption

Recall that our main problem is to find if any term t a large set T of indexed terms subsumes a given query term q. Using compilation described in the previous subsection, one can solve the problem by the execution of code for all terms in T. This solution is inapropriate for large sets of terms. However, code sequences for terms can still be useful as we can share many instructions from code for different terms. We rely on the following observation: in most instances in automated theorem proving the set T contains many terms having similar structure. Code sequences for similar terms often have long coinciding prefixes. It is natural to combine the code sequences into one indexing structure, where the equal prefixes of code sequences are shared. Due to the tree-like form of such structures we call them *code trees*. Nodes of code trees are instructions of the abstract subsumption machine. Linking of different code sequences is done by setting appropriate values to the *cont* and *back* arguments of the instructions. A branch of such tree is a code sequence for some indexed term interleaved by some instructions of code sequences for other indexed terms. Apart from reducing memory consumption, combining code sequences in one index results in tremendous improvements in time-efficiency since during a subsumption check shared instructions are executed once for several terms in the indexed set. To illustrate this idea let us compare the code sequences for the terms $t_1 = f(f(x_1, x_2), f(x_1, x_1))$ and $t_2 = f(f(x_1, x_2), f(x_2, x_2))$.

$$
\begin{array}{ll}
\mathit{initl} : \mathit{Initialize}(l_1) & \mathit{initl} : \mathit{Initialize}(l_1) \\
l_1 : \mathit{Check}(f, l_2, \mathit{faill}) & l_1 : \mathit{Check}(f, l_2, \mathit{faill}) \\
l_2 : \mathit{Check}(f, l_3, \mathit{faill}) & l_2 : \mathit{Check}(f, l_3, \mathit{faill}) \\
l_3 : \mathit{Put}(1, l_4, \mathit{faill}) & l_3 : \mathit{Put}(1, l_4, \mathit{faill}) \\
l_4 : \mathit{Put}(2, l_5, \mathit{faill}) & l_4 : \mathit{Put}(2, l_5, \mathit{faill}) \\
l_5 : \mathit{Check}(f, l_6, \mathit{faill}) & l_5 : \mathit{Check}(f, l_6, \mathit{faill}) \\
l_6 : \mathit{Put}(3, l_7, \mathit{faill}) & l_6 : \mathit{Put}(3, l_7, \mathit{faill}) \\
l_7 : \mathit{Put}(4, l_8, \mathit{faill}) & l_7 : \mathit{Put}(4, l_8, \mathit{faill}) \\
l_8 : \mathit{Compare}(1, 3, l_9, \mathit{faill}) & l_8 : \mathit{Compare}(2, 3, l_9, \mathit{faill}) \\
l_9 : \mathit{Compare}(1, 4, l_{10}, \mathit{faill}) & l_9 : \mathit{Compare}(2, 4, l_{10}, \mathit{faill}) \\
l_{10} : \mathit{Success} & l_{10} : \mathit{Success}
\end{array}
$$

Sharing the first eight instructions of this results in the following code:

$$
\begin{array}{ll}
\quad C : & \\
\mathit{initl} : \mathit{Initialize}(l_1) & \\
l_1 : \mathit{Check}(f, l_2, \mathit{faill}) & \\
l_2 : \mathit{Check}(f, l_3, \mathit{faill}) & \\
l_3 : \mathit{Put}(1, l_4, \mathit{faill}) & \\
l_4 : \mathit{Put}(2, l_5, \mathit{faill}) & \\
l_5 : \mathit{Check}(f, l_6, \mathit{faill}) & \\
l_6 : \mathit{Put}(3, l_7, \mathit{faill}) & \\
l_7 : \mathit{Put}(4, l_8, \mathit{faill}) & \\
l_8 : \mathit{Compare}(1, 3, l_9, l_{11}) & l_{11} : \mathit{Compare}(2, 3, l_9, \mathit{faill}) \\
l_9 : \mathit{Compare}(1, 4, l_{10}, \mathit{faill}) & l_{12} : \mathit{Compare}(2, 4, l_{10}, \mathit{faill}) \\
l_{10} : \mathit{Success} &
\end{array}
$$

Control flow of this code, depicted on Figure 7, has form of a tree. This explains the choice of the term "code tree".

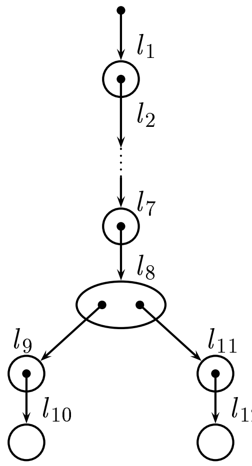

Figure 7: Control flow of the code C

We can execute this code as follows. First, the eight shared instructions are executed. If none of them results in failure, we continue by executing instructions l_8, l_9, l_{10}. If the *Success* instruction l_{10} is reached the whole process terminates with success. Otherwise, if any of the equality checks l_8, l_9, failed, we have to backtrack and resume the execution from the instruction l_{11}.

In general, to maintain a code tree for a dynamicaly changing set T, one has to implement two operations: integration of new code sequences into the tree, when a term is added to T, and removal of sequences when a term is deleted from T. The integration of a code sequence CS into a code tree CT can be done as folows. We move simultaniously along the sequence CS and a branch of CT beginning from the *Initialize* instructions. If the current instruction I_T in CT coincides with the current instruction I_S in CS up to the label arguments, we skip the instructions following labels in their *cont* arguments. If I_T differs from I_S we have to consider two cases:

1. If $back(I_T)$ is not the *Failure* instruction, in the code tree we move to this instruction and continue integration.

2. If $back(I_T)$ is *Failure*, we set the *back* argument of I_T to the label of I_S. Thus, the rest of the code sequence CS together with the passed instructions in CT forms a new branch in the tree.

Removal of obsolete branches is also very simple: we remove from the code all unshared instructions corresponding to the removed term and link the remaining instructiions in appropriate manner. Due to postponing *Compare* instructions, code trees maitained in this manner have an important property: traversal codes for any terms having the same technical skeleton are shared completely.

Code trees are executed nearly the same way as code sequences, but with one difference due to possible backtrack points. As soon as an instruction with a backtrack argument has been found, we store its backtrack argument and the current position in the query term in special stacks *backtrPos* and *backtrInstr*. Semantics of instructions in code trees is shown in Figure 8

It is worth noting that all operations in the semantics of instructions can be executed very efficiently on flatterms.

Initialize(l_1)	$p := \lambda$; $backtrPos$:= *empty stack*; $backtrInstr$:= *empty stack*; **goto** l_1	*Check*(s, l_1, l_2)	**if** $top(q/p) = s$ **then** $push(l_2, backtrInst)$; $push(p, backtrPos)$; $p := next_q(p)$; **goto** l_1 **else goto** l_2
Put(n, l_1, l_2)	$push(l_2, backtrInst)$; $push(p, backtrPos)$; $subst[n] := p$; $p := after_q(p)$; **goto** l_1	*Compare*(m, n, l_1, l_2)	**if** $q/subst[m] = q/subst[n]$ **then** $push(l_2, backtrInst)$; $push(p, backtrPos)$; **goto** l_1 **else goto** l_2
Success	**return** success	*Failure*	**if** *backtrPos* is empty **then return** failure **else** $p = pop(backtrPos)$; **goto** $pop(backtrInst)$

Figure 8: Semantics of instructions in code trees

To conclude the section we descibe here the differences between this version of code trees and that of [Voronkov 1995]. These differences make the execution of code trees significantly faster:

1. The original version of code trees contained 6 more instructions:

   (a) The flatterm representation of queries made it possible to get rid of the stack instructions *Push* and *Pop* heavily used in the original version to encode term-traversal related operations.

   (b) Effect of the *Right* and *Down* instructions is now part of the semantics for *Check* and *Put*. This saves space and time: instead of fetching two instructions we only need to fetch one (instructions are interpreted, so there is an overhead in fetching the next instruction).

   (c) Due to better organization of backtracking, the instructions *Fork* and *Restore* used for the maintanence of backtracking are not needed any more.

2. The execution of all instructions except *Compare* requires constant time. The most expensive *Compare* instructions require comparison of two subterms of the query term. Due to the flatterm representation of the query term, *Compare* instructions are now executed more efficiently.

4 Partially adaptive code trees

From the discussion in the previous section the reader could get a feeling that code trees are slightly optimized discrimination trees. In this section we discuss an optimization which is essentially impossible on discrimination trees. This optimization, *partially adaptive code trees*, shows greater flexibility of code trees as compared to discrimination trees.

It is believed that a greater amount of sharing, and hence efficiency, can be gained by using *adaptive* indexing structures [see Ramakrishnan et al. 2000]. An example of such structure is *substitution trees* [Graf 1995] or *adaptive automata* [see Ramakrishnan et al. 2000].

The idea of adaptive structures is that the order of the query term traversal is not fixed in advance, so indexing structures can adapt themselves to new orders of traversal when query terms are added or deleted. The price paid for adaptiveness is quite high, so it is not clear that adaptive structures can be more efficient than the standard ones. The index maintainance becomes more complex, choosing a wrong order can actually slow down execution, and it is difficult to ensure that the order is good: usually, the problem of optimality of a given structure is coNP-complete [see Ramakrishnan et al. 2000]. In the case of code trees for forward subsumption, the use of adaptive structures requires tree-like representation of query terms and, consequently, a larger set of instructions.

However, the flexibility of code trees allows one to make them *partially adaptive*, without changing the order of traversal of query terms. The main idea is to use the fact that *Compare* instructions commute with many other instructions, and thus can be moved up and down the tree (with essentially no overhead in the index maintainence). To illustrate this idea, consider the term $t_1 = f(x_1, x_1, x_2, x_2)$ and the following code sequences C_1, C_1':

C_1 :	C_1' :
$initl : Initialize(l_1)$	$initl : Initialize(l_1)$
$l_1 : Check(f, l_2, faill)$	$l_1 : Check(f, l_2, faill)$
$l_2 : Put(1, l_3, faill)$	$l_2 : Put(1, l_3, faill)$
$l_3 : Put(2, l_4, faill)$	$l_3 : Put(2, l_4, faill)$
$l_4 : Put(3, l_5, faill)$	$l_4 : Compare(1, 2, l_5, faill)$
$l_5 : Put(4, l_6, faill)$	$l_5 : Put(3, l_6, faill)$
$l_6 : Compare(1, 2, l_7, faill)$	$l_6 : Put(4, l_7, faill)$
$l_7 : Compare(3, 4, l_8, faill)$	$l_7 : Compare(3, 4, l_8, faill)$
$l_8 : Success$	$l_8 : Success$

The code sequence C_1 is computed by our compilation algorithm. The code sequence C_1' is obtained from C_1 by moving the instruction $Compare(1, 2, \ldots)$ up the sequence. Such a lifting of some *Compare* instructions serves two purposes. The first one is earlier detection of failure. For example, execution of the code C_1 on the query term $q = f(a, b, a, a)$ determines failure after 7 instructions, while C_1' fails after 5 instructions.

The second purpose of moving instructions up the tree is that it can increase sharing of code when new code sequences are integrated into code trees. Moreover, since *Compare* are potentially expensive instructions, sharing of them is especially desirable. For example, consider the term $t_2 = f(x_1, x_1, a, x_2)$ and two equivalent code sequences for t_2:

C_2 :	C'_2 :
$initl : Initialize(l_1)$	$initl : Initialize(l_1)$
$l_1 : Check(f, l_2, faill)$	$l_1 : Check(f, l_2, faill)$
$l_2 : Put(1, l_3, faill)$	$l_2 : Put(1, l_3, faill)$
$l_3 : Put(2, l_4, faill)$	$l_3 : Put(2, l_4, faill)$
$l_4 : Check(a, l_5, faill)$	$l_4 : Compare(1, 2, l_5, faill)$
$l_5 : Put(3, l_6, faill)$	$l_5 : Check(a, l_6, faill)$
$l_6 : Compare(1, 2, l_7, faill)$	$l_6 : Put(3, l_7, faill)$
$l_7 : Success$	$l_7 : Success$

Combining C_1 with C_2 gives us the following code tree:

T :	
$initl : Initialize(l_1)$	
$l_1 : Check(f, l_2, faill)$	
$l_2 : Put(1, l_3, faill)$	
$l_3 : Put(2, l_4, faill)$	
$l_4 : Put(3, l_5, l_9)$	$l_9 : Check(a, l_{10}, faill)$
$l_5 : Put(4, l_6, faill)$	$l_{10} : Put(2, l_8, faill)$
$l_6 : Compare(1, 2, l_7, faill)$	$l_{11} : Compare(1, 2, l_8, faill)$
$l_7 : Compare(3, 4, l_8, faill)$	
$l_8 : Success$	

Combining C'_1 with C'_2 gives us a code tree with less instructions:

T' :	
$initl : Initialize(l_1)$	
$l_1 : Check(f, l_2, faill)$	
$l_2 : Put(1, l_3, faill)$	
$l_3 : Put(2, l_4, faill)$	
$l_4 : Compare(1, 2, l_5, faill)$	
$l_5 : Put(3, l_6, l_9)$	$l_9 : Check(a, l_{10}, faill)$
$l_6 : Put(4, l_7, faill)$	$l_{10} : Put(2, l_8, faill)$
$l_7 : Compare(3, 4, l_8, faill)$	
$l_8 : Success$	

Execution of the code tree T on the query term $f(a, b, a, a)$ fails after 10 instructions, while execution of T' fails only after 5.

Under some circumstances, *Compare* instructions can also be moved down the tree, for the same purpose of increasing sharing. We will illustrate this later, when we discuss the algorithm of insertion into code trees. Thus, the new code trees can adapt themselves to the insertion of new code sequences by moving some instructions up and down the tree (but without changing the order of traversal of the query term). This is why we call them *partially adaptive*.

Apart from moving *Compare* instructions, other equivalence-preserving transformations of code sequences can be used to improve sharing. This optimization is based on the observation that different computation sequences can be used for computing an equivalence relation. When encoding the technical equivalence $\mathcal{E}_t$ by a sequence of *Compare* instructions we can use any computation sequence for $\mathcal{E}_t$ instead of the canonical one.

EXAMPLE 4.1 Let us illustrate this idea by an example. Consider the terms $t_1 = f(x_1, x_2, x_2, x_2)$ and $t_2 = f(x_1, x_1, x_1, x_2)$. The canonical computation sequences for $\mathcal{E}_{t_1}$ and $\mathcal{E}_{t_2}$ are $< 2, 3 >, < 2, 4 >$ and $< 1, 2 >, < 1, 3 >$. The correponding *Compare* instructions in the code sequences for t_1 and t_2 cannot be shared. However, the equivalence relation $\mathcal{E}_{t_2}$ can be computed by the sequence $< 2, 3 >, < 1, 3 >$, so that the instructions $\mathit{Compare}(2, 3, \ldots)$ can be shared resulting in the following code tree for $\{t_1, t_2\}$:

$$
\begin{array}{ll}
\mathit{initl} : \mathit{Initialize}(l_1) & \\
l_1 : \mathit{Check}(f, l_2, \mathit{faill}) & \\
l_2 : \mathit{Put}(1, l_3, \mathit{faill}) & \\
l_3 : \mathit{Put}(2, l_4, \mathit{faill}) & \\
l_4 : \mathit{Put}(3, l_5, \mathit{faill}) & \\
l_5 : \mathit{Compare}(2, 3, l_6, \mathit{faill}) & \\
l_6 : \mathit{Put}(4, l_7, l_9) & \\
l_7 : \mathit{Compare}(3, 4, l_8, \mathit{faill}) & l_9 : \mathit{Compare}(1, 3, l_8, \mathit{faill}) \\
l_8 : \mathit{Success} &
\end{array}
$$

Note that the semantics of instructions in partially adaptive code trees is the same as in the standard code trees. The only difference between the two versions of code trees is in their maintenance: the compilation of code sequences and their insertion into a code tree.

Now specialising the algorithm on a given term may produce several different codes. We have to fix a strategy of chosing an appropriate code sequence for a given term in presence of a code tree. The choice of the strategy must reflect our two main goals: better degree of sharing and earlier detection of failure. Moreover, we often have to modify the tree itself significantly since some code sequences in the tree are to be adapted to the new code sequences being integrated. Thus, the situation is more complex than with the basic version, compilation should be done simultaneously with modifying the tree. Our third goal is efficiency of maintainence: the insert into and deletion from code trees should be fast.

In view of the third goal, the deletion algorithm we use in Vampire is very simple. After having deleted a code sequence from a tree we do not try to modify the trees by shifting *Compare* instructions. This means that the code tree for a set of indexed terms T can change when we insert a code sequence for a new indexed term t, and then immediately delete this code sequence.

We will now focus on the algorithm for insertion into code trees. We do not define the algorithm here, but only describe it informally and give an illustrating example. The algorithm is similar to the standard insertion algorithm into code trees (or discrimination trees), but with the following difference. First, we make insertion by ignoring *Compare* instructions at all. Second, we shift some *Compare* instructions down the tree. Third, we insert remaining *Compare* instructions from the new code.

EXAMPLE 4.2 Consider the following code tree for the set $\{f(x_1, x_1, x_2, x_2), f(x_1, x_1, a, b)\}$:

$$
\begin{array}{ll}
\mathit{initl} : \mathit{Initialize}(l_1) & \\
l_1 : \mathit{Check}(f, l_2, \mathit{faill}) & \\
l_2 : \mathit{Put}(1, l_3, \mathit{faill}) & \\
l_3 : \mathit{Put}(2, l_4, \mathit{faill}) & \\
l_4 : \mathit{Compare}(1, 2, l_5, \mathit{faill}) & \\
l_5 : \mathit{Put}(3, l_6, l_9) & l_9 : \mathit{Check}(a, l_{10}, \mathit{faill})
\end{array}
$$

$l_6 : Put(4, l_7, faill)$ $\quad$ $l_{10} : Check(b, l_8, faill)$
$l_7 : Compare(3, 4, l_8, faill)$
$l_8 : Success$

Suppose that we insert into the set the new term $t = f(x_1, x_2, x_1, x_1)$. The code sequence for this term consists of the traversal code

$initl : Initialize(m_1)$
$m_1 : Check(f, l_2, faill)$
$m_2 : Put(1, l_3, faill)$
$m_3 : Put(2, l_4, faill)$
$m_4 : Put(3, l_5, faill)$
$m_5 : Put(4, l_6, faill)$

followed by a sequence of *Compare* corresponding to a computation sequence for the equivalence relation $\mathcal{E}_t$ consising of two classes $\{1, 3, 4\}$ and $\{2\}$.

If we ignore the *Compare* instructions in the code tree, then the nodes m_1, m_2, m_3, m_4, m_5 would be merged into the nodes l_1, l_2, l_3, l_5, l_6, respectively. But between l_3 and l_4 the tree contains the instruction $l_4 : Compare(1, 2, l_5, faill)$, and $< 1, 2 >$ does not belong to $\mathcal{E}_t$. So, we have to move $Compare(1, 2, l_5, faill)$ down the tree. The instruction $l_7 : Compare(3, 4, l_8, faill)$ can be shared, since $< 3, 4 >$ belongs to $\mathcal{E}_t$. To compute the equivalence relation $\mathcal{E}_t$, we should add either $< 1, 3 >$ or $< 1, 4 >$ to the computation sequence $< 3, 4 >$. So, we obtain the following code tree:

$initl : Initialize(l_1)$
$l_1 : Check(f, l_2, faill)$
$l_2 : Put(1, l_3, faill)$
$l_3 : Put(2, l_5, faill)$
$l_5 : Put(3, l_6, l_{11})$ $\quad$ $l_{11} : Compare(1, 2, l_9, faill)$
$l_6 : Put(4, l_7, faill)$ $\quad$ $l_9 : Check(a, l_{10}, faill)$
$l_7 : Compare(3, 4, l_4, faill)$ $\quad$ $l_{10} : Check(b, l_8, faill)$
$l_4 : Compare(1, 2, l_8, l_{12})$ $\quad$ $l_{12} : Compare(1, 3, l_8, faill)$
$l_8 : Success$

5 Further optimizations

In this section we consider several "optimizations" we tried to implement in code trees. For all of them, the results were negative: their implementation gave no essential improvement, or even resulted in a stable slow-down. Two of them: signature and weight filters, are implemented in some other provers, for example, weight filters corresponds to *size constraints* added to discrimination trees in the prover *E* [Schulz 1999].

The fact that we obtained no speedup by using these filters shows high efficiency of code trees as compared to discrimination trees.

Tabulation of *Compare* results. One must realise that the advantages achieved by earlier equality checks often come at the cost of executing relatively costly *Compare* instructions before some *Check* instructions that may detect failure. This negative effect may sometimes result in decline of the overall perfomance.

We can tabulate the results of the comparison in a special array *comp* whose elements are members of the set $\{equal, not\_equal, unknown\}$. When we need to compare terms at positions p_1, p_2 in the query term, we inspect the value of $comp(p_1, p_2)$. If such a comparison was already done, the value is either *equal* or *not_equal*, and we simply use this value. Otherwise, we perform the comparison and tabulate the result in $comp(p_1, p_2)$. The use of tabulation resulted in a slowdown over nearly all TPTP problems with some rare exceptions.

Signature filters. We extended the set of instructions of the abstract machine by the *signature filter* instructions. These instructions contain information about the function symbols that occur in *every* branch below a given node in the code tree. This information is used to check against the signature of the query term. If every branch below the current node contains a symbol that does not occur in the query term, then no branch below the current node can lead to success. Despite a very optimized implementation of signature filters (for example, there is no signature filter for f if the instruction $Check(f, \ldots)$ is in the next node), we did not get any improvement over the standard or partially adaptive code trees.

Weight filters. Likewise, we tried to use weight filters for subsumption by unit clauses. The idea is that a unit clause cannot subsume a clause with a smaller weight. Again, there was no improvement over the standard or partially adaptive code trees.

6 Experiments

Our experiments have shown that in many cases making code trees partially adaptive gives significant reduction of the total number of executed instructions, though at the cost of increasing the share of *Compare* instructions. We compared overall performance of the system with the partialy adaptive version of code trees and the basic version. The problems were run with the time limit of 10 minutes on a PC with two Pentium III 500MHz processors. We restricted memory usage by 300Mb limit and the number of kept clauses was limited by 100000. In the table below we give times consumed by the optimised version and the basic one ($time_o$ and $time_b$ correspondingly). To make comparison, we calculate percentage of difference (*diff*) between times consumed by the versions w.r.t. the best time. Negative value of *diff* indicates cases when the optimised version showed worse results. From the whole benchmark suit 34 problems were selected by the following criteria: (1) one of the versions works at least 30 seconds on the selected problems with the given limits, (2) to make fair comparison of the overall performance, both versions must show same behaviour (up to timing) on a selected problem, (3) absolute value of *diff* must exceed 1%

problem	*diff*	$time_o$	$time_b$	*problem*	*diff*	$time_o$	$time_b$
alg003-1	3.33%	41.13	42.5	lcl005-1	-1.4%	92.45	91.17
alg004-1	9.86%	37.9	41.64	lcl015-1	-1.26%	72.09	71.19
boo020-1	-3.05%	42.2	40.95	lcl016-1	-1.27%	71.27	70.37
cid003-1	9.92%	35.67	39.21	lcl017-1	2.47%	72.05	73.83
cid003-2	6.36%	46.97	49.96	lcl020-1	10.74%	77.41	85.73
civ002-1	-2.54%	86.59	84.44	lcl021-1	5.28%	76.89	80.95
col077-1	-1.38%	30.79	30.37	lcl099-1	4.62%	42.81	44.79
grp054-1	1.48%	51.24	52	lcl105-1	4.31%	39.15	40.85
grp073-1	-1.39%	34.22	33.75	lcl122-1	1.14%	80.34	81.26
grp106-1	-1.19%	47.61	47.05	lcl125-1	-1.01%	36	35.64
grp107-1	-1.11%	70.55	69.77	lcl127-1	3.99%	57.02	59.3
grp108-1	-1.15%	56.93	56.28	lcl129-1	3.7%	32.42	33.61
grp110-1	-1.54%	40.2	39.59	lcl166-1	5.54%	77.36	81.65
grp111-1	-1.1%	53.86	53.27	lcl167-1	10.48%	77.26	85.36
lat002-1	-2.53%	61.13	59.62	prv008-1	9.65%	166.47	182.55
lat005-3	-1.75%	61.31	60.25	rng025-1	49.18%	31.98	47.71
lat005-4	-4.15%	60.43	58.02	rng034-1	11.46%	50.69	56.5

References

Christian J. [1993], 'Flatterms, discrimination nets, and fast term rewriting', *Journal of Automated Reasoning* **10**(1), 95–113.

Graf P. [1995], Substitution tree indexing, *in* J. Hsiang, ed., 'Rewriting Techniques and Applications', Vol. 914 of *Lecture Notes in Computer Science*, pp. 117–131.

Graf P. [1996], *Term Indexing*, Vol. 1053 of *Lecture Notes in Computer Science*, Springer Verlag.

McCune W. W. [1992], 'Experiments with discrimination-tree indexing and path indexing for term retrieval', *Journal of Automated Reasoning* **9**(2), 147–167.

Ramakrishnan I., Sekar R. and Voronkov A. [2000], Term indexing, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Elsevier Science and MIT Press. To appear.

Ryazanov A. and Voronkov A. [1999], Vampire, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Lecture Notes in Artificial Intelligence, Trento, Italy, pp. 292–296.

Schulz S. [1999], System abstract: E 0.3, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Lecture Notes in Artificial Intelligence, Trento, Italy, pp. 297–301.

Sutcliffe G. [2000], 'The CADE-16 ATP system competition', *Journal of Automated Reasoning* . to appear.

Voronkov A. [1995], 'The anatomy of Vampire: Implementing bottom-up procedures with code trees', *Journal of Automated Reasoning* **15**(2), 237–265.

Index